

# PCS: Predictive Component-level Scheduling for Reducing Tail Latency in Cloud Online Services

Rui Han, Junwei Wang, Siguang Huang, Chenrong Shao, Shulin Zhan, Jianfeng Zhan  
*Institute Of Computing Technology,  
 Chinese Academy of Sciences  
 Beijing, China*  
 hanrui,wangjunwei,huangsiguang,shaochenrong,zhanshulin,zhanjianfeng@ict.ac.cn

Jose Luis Vazquez-Poletti  
*Facultad de Informatica,  
 Universidad Complutense de Madrid  
 Madrid, Spain*  
 jlvazquez@fdi.ucm.es

**Abstract**—Modern latency-critical online services often rely on composing results from a large number of server components. Hence the tail latency (e.g. the 99th percentile of response time), rather than the average, of these components determines the overall service performance. When hosted on a cloud environment, the components of a service typically co-locate with short batch jobs to increase machine utilizations, and share and contend resources such as caches and I/O bandwidths with them. The highly dynamic nature of batch jobs in terms of their workload types and input sizes causes continuously changing performance interference to individual components, hence leading to their latency variability and high tail latency. However, existing techniques either ignore such fine-grained component latency variability when managing service performance, or rely on executing redundant requests to reduce the tail latency, which adversely deteriorate the service performance when load gets heavier. In this paper, we propose PCS, a predictive and component-level scheduling framework to reduce tail latency for large-scale, parallel online services. It uses an analytical performance model to simultaneously predict the component latency and the overall service performance on different nodes. Based on the predicted performance, the scheduler identifies straggling components and conducts near-optimal component-node allocations to adapt to the changing performance interferences from batch jobs. We demonstrate that, using realistic workloads, the proposed scheduler reduces the component tail latency by an average of 67.05% and the average overall service latency by 64.16% compared with the state-of-the-art techniques on reducing tail latency.

**Keywords**—cloud online services; component latency variability; tail latency; predictive scheduler;

## I. INTRODUCTION

Providing fluid responsiveness to user requests is essential for online services: their potential profits are proportional to service latency (i.e. request response time including both the request queueing delay and the time of being processed) [14], [18]. In large online services such as search engines, e-commerce sites, and social networks, the processing of incoming requests consists of several sequential stages, where each stage composes responses parallelized across hundreds or thousands of server *components*. Hence the tail (e.g. the 99th percentile) of these components' latencies, rather than the average, determines the overall service performance

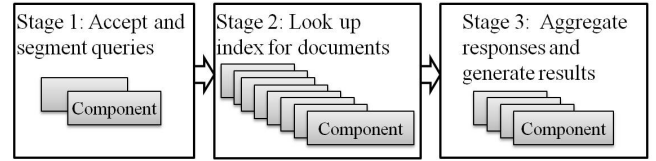


Figure 1. An example of Nutch search engine

[14], [21]. For example, Figure 1 shows an example Nutch search engine [1] with three stages. Suppose that at stage 2, the request processing is parallelized into 100 components, in which 99 components can respond in 10ms but only one component gets a slow response of 1 second, the overall service performance is deteriorated by this straggling component and hence providing slow responsiveness of 1 second.

In modern cloud data centers and warehouse-scale computers, it is critical to improve machine utilizations by co-locating long-running online services and offline batch jobs (e.g. Hadoop [5] and Spark [8] analytics jobs) on the same node (physical machine), while still keeping the overall latency of online services at a satisfactory level [24], [31]. Although the components of a service are typically hosted on dedicated environments such as Xen virtual machines (VMs) or Linux Containers (LXCs), these components still share and contend resources such as processing units, caches and I/O bandwidths with their co-running batch jobs on the same node, hence inevitably suffer from performance interference. Workload traces from Google [24] and Facebook [13] show that small batch jobs form a majority (over 90%) of all jobs in their data center workloads. For example, approximately 50% of Google jobs complete in 10 minutes and 94% of them complete within 3 hours. These short-term batch jobs have various workload types (e.g. CPU and I/O intensive workloads) and input data sizes (e.g. ranging from KB to GB), thus causing continuously changing performance interferences to their co-located components. This results in the **component latency variability**, which can be explained from two aspects: (a) each component's latency

(performance) varies over time, and (b) components hosted on different nodes have different changes in their latencies, hence causing high tail latency in individual components of the service.

Many existing techniques have been developed to guarantee the performance of latency-critical services by mitigating the performance interference due to resource sharing and contention [10], [22], [29], [30]. However, these techniques only manage service performance at the coarse granularity of the entire application, ignoring fine-grained component latency variability that may come to dominate service performance at large scale. Moreover, state-of-the-art techniques reduce tail latency via request redundancy. They either create replicas for all the requests [11], [26], [27] or reissue slow requests' replicas to a different component [14], [18], and then use the quickest replica. Although these techniques work well under light load, they adversely deteriorate the service performance when load gets heavier [25].

In this paper, we propose a new component-level service scheduler that dynamically schedules the components of a service to appropriate nodes with the assistance of cost-effective online monitors. Compared to existing latency reduction techniques, the proposed scheduler applies an analytic performance model to predict the latencies of all components and their impact on the overall service performance, and then formulates the scheduling decisions based on the predicted performance. The performance model also dynamically updates the prediction results at each scheduling interval by collecting the latest resource contention information during the service execution, thus allowing the scheduler to adapt to changes in performance interference. The concrete contributions of this work are as follows:

- We build a flexible analytic performance model to accurately predict the performance of an online service. The basic model comprehensively covers some of the most representative shared resources that are likely to incur contentions and predicts each component's service time on different nodes by taking the resource contention and performance interference into consideration. The extended model further considers the request queueing delay and estimates the latency of the whole service based on its implementation topology. We show that the proposed model can predict the latency with an average error of 2.68%.
- Based on the performance model, we present a framework for component-level scheduling. At each scheduling interval, our approach efficiently identifies the straggling components of a service such that the migration of these components brings the maximum reduction in the overall service latency. The effectiveness of the proposed approach is evaluated using comparative experiments on a variety of realistic workloads publicly available from the BigDataBench suite [3]. The experiment results in a 100-machine cluster demonstrate that

compared with the state-of-the-art techniques on mitigating tail latency, our approach reduces components' 99th percentile latency by an average of 67.05% and the average overall service latency by 64.16%.

The remainder of this paper is organized as follows: Section II introduces the background information. Section III gives an overview of the proposed scheduling framework. Section IV presents the performance model and Section V explains the scheduling algorithm. Section VI evaluates the proposed approach. Section VII presents the related work, and finally, Sections VIII summarizes the work.

## II. BACKGROUND

### A. Sources of component latency variability

*Resource sharing and contention.* When deploying an online service on a cloud platform, the performance interference due to the co-located batch jobs' resource contention is often regarded as a major cause of a component's service time variability [14], [23]. Some system activities including hardware activities (such as garbage collections of storage devices and energy management behaviors) and software activities (such as kernel daemons and system maintenance) also influence the component's service time.

*Queueing delay.* The component's service time variability is significantly amplified in the request queueing delay when considering different request arrival rates. Hence the variability of service time and queueing delay work together to cause large latency variability in individual components.

### B. Dynamic performance interference of batch jobs

The dynamic performance interference of batch jobs are caused by their short running periods and continually changing workload characteristics, which can be explained in two aspects.

*Workload type.* It has twofold meanings: (i) *Computation semantics.* Batch jobs with different computation semantics (i.e. source codes) may have different resource demands. For example, Sort is an I/O-intensive workload, Bayes classification is a CPU-intensive workload with dominated floating point operations, and Page Index has similar demands for CPU and I/O resources. (ii) *Software stacks.* Model software stacks such as Hadoop and Spark usually provide rich libraries to facilitate development of new applications, and allow a programmer focus on writing a few lines of codes to implement an application. Hence a batch job of the same computation semantic may have considerably different resource demands when implemented with different software stacks [20]. For example, Hadoop Bayes is a CPU-intensive workload but Spark Bayes is an I/O-intensive workload.

*Input data size.* The resource demand of a job varies when it processes different input data sizes. For example, when running on a 12-core Xeon E5635 processor, the CPU utilizations of the WordCount workload are 31%, 61%, and

79% when its input data sizes are 500MB, 2GB, and 8GB, respectively.

### III. OVERVIEW OF THE FRAMEWORK

As shown in Figure 2, the proposed framework for predictive component-level scheduling consists of three modules: the on-line monitors, the performance predictor and the scheduling heuristic.

The *on-line monitor* continuously detects two types of information in a running service, whose components are distributed on  $k$  nodes of a data center. The first type of information represents the service's workload status, i.e. its request arrival rate. The second type of information reflects the resource contention information of each component due to its co-located programs on the same node. Specifically, the monitor obtains the request arrival rate by profiling service's running logs, collects system-level contention information (e.g. core usage and I/O bandwidths) by accessing the proc file system, and profiles micro-architectural contention information (e.g. shared cache misses) using hardware performance counters for Linux 2.6+ based systems. In our monitor, Perf [7] is used to profile physical machines and Oprofile [6] is used to profile VMs.

At the end of each scheduling interval, the *performance predictor* collects the monitored information and predicts the component's latency on all  $k$  nodes. This predictor also estimates the impact of individual component latencies on the performance of the whole service based on its implementation topology, and organizes the predicted values as a performance matrix. Using this matrix, the *data center scheduler* applies the scheduling algorithm to identify the straggling components and enforces the appropriate node assignment of the components for the next interval. Consequently, the framework is able to dynamically and efficiently adapt to component latency variability.

Note that the proposed scheduling algorithm is not intended to replace, but rather complement the existing scaling or resource provisioning techniques (e.g. reactive scaling up [16] or prediction-based resource provisioning [12], [15] approaches) for multi-stage online services. Specifically, the component-level scheduling is enforced only after the machines have been allocated to the service. At each scheduling interval, the component-node allocation can be conducted by calling the deployment APIs offered by existing distributed realtime computation systems such as Storm [2] and Drill [4] to migrate the components to the available machines (e.g. VMs or LXCs) on the scheduled nodes. Note also that although this component-node allocation can be enforced by directly migrating the machines to the nodes, we prefer the former solution as it produces lower overheads on scheduling.

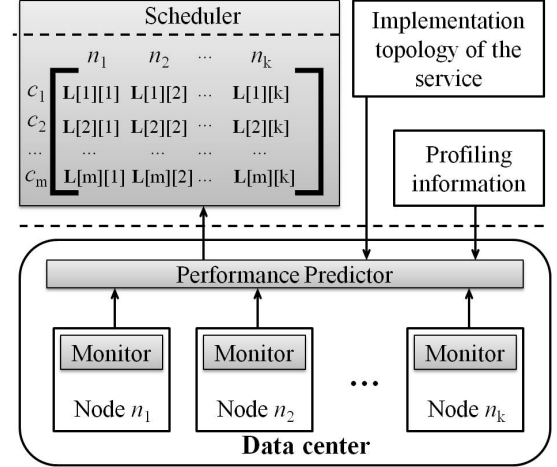


Figure 2. The overview of the framework

### IV. PERFORMANCE PREDICTOR

Predicting a component's latency when running on different nodes is the key step to detect straggling components in a service. This requires the performance predictor to consider all causes of latency variability discussed in Section II-A. In the presence of fine-grained heterogeneity of resource contentions on each component, the basic performance predictor is responsible for collecting the information of resource sharing and contention, and predicting the impact on individual component's performance (Section IV-A). The extended performance model further estimates the component's latency by taking the current request arrival rate into account, and calculates the overall service latency based on the service implementation topology (Section IV-B). With these two models, the performance predictor finally exposes the component latency variability to the scheduler as a performance matrix of reduced overall service latencies (Section IV-C). Table I lists all notations.

Table I  
TABLE OF NOTATIONS

Symbol	Meaning
$n$	A node
$c$	A component belonging to a service
$x$	$c$ 's service time
$sr$	One type of shared resources
$U_{sr}$	$sr$ 's resource contention information
$\mathbf{U}$	The contention vector consists of contention information of all shared resources
$RG(U_{sr})$	A basic regression model
$RG_{ST}(\mathbf{U})$	A combined regression model representing the predicted service time
$l$	$c$ 's latency
$l_{stage}$	A stage's latency
$l_{overall}$	The overall service latency
$\mathbf{L}$	The matrix of the reduced overall service latency
$\mathbf{L}[i][j]$	An entry of $\mathbf{L}$ , which represents the reduced overall latency when component $c_i$ 's is migrated from its current node to node $n_j$

### A. Basic performance model

Given a component  $c$  hosted on a node  $n$ , the basic performance model is developed to capture the impact of resource sharing and contention on  $c$ 's performance and estimate its service time  $x$ .

Table II lists the contention information of shared resources. The model comprehensively considers both on-chip resources (e.g. shared processing units and caches) and off-chip resources (disk and network bandwidths) contended by different programs on node  $n$ . In Table II, core usage represents the ratio of time running instructions on the cores (including private cache hits); MPKI represents the number of instruction Misses Per Kilo Instructions of shared caches including last level cache (LLC), instruction Translation Lookaside Buffer (TLB/ITLB), and data TLB (DTLB). MPKI thus indicates the stalled cycles due to cache contention. Note that the *contention* of these resources comes from  $c$ 's co-running programs within the same service or across other applications, and node  $n$ 's hardware/software activities.

Table II  
CONTENTION INFORMATION OF SHARED RESOURCES

Shared resources	Contention information
Floating point and vector processing units, pipelines, and data prefetchers	$U_{core}$ =core usage
LLC, ITLB, DTLB	$U_{cache}$ =MPKI
Disk bandwidth	$U_{diskBW}$ =the amount of read/write data per second
Network bandwidth	$U_{networkBW}$ =the amount of send/receive data per second

Based on the contention information, the basic performance model predicts  $c$ 's service time  $x$  using two steps. The first step employs a regression model to describe the relationship between one contention information and  $c$ 's service time. The training of the regression model takes a set of  $v$  samples  $\{(U_{sr_1}, x_1), \dots, (U_{sr_v}, x_v)\}$  as input and outputs a model  $RG(U_{sr})$ , where  $sr \in \{core, cache, diskBW, networkBW\}$  and  $U_{sr_i} \in \{U_{core_i}, U_{cache_i}, U_{diskBW_i}, U_{networkBW_i}\}$  ( $i=1, \dots, v$ ). Hence  $c$ 's service time  $x$  is predicted as  $RG(U_{sr})$  when the contention information is  $U_{sr}$ . The training samples are obtained from profiling runs or historical running logs.

During the training of regression model, the first step also calculates the relevance (i.e. weight  $w_{sr}$ ) between the contention information of shared resource  $sr$  and  $c$ 's service time. Suppose four regression models ( $RG_{U_{core}}$ ,  $RG_{U_{cache}}$ ,  $RG_{U_{diskBW}}$  and  $RG_{U_{networkBW}}$ ) and their weights ( $w_{core}$ ,  $w_{cache}$ ,  $w_{diskBW}$  and  $w_{networkBW}$ ) are obtained, the second step predicts  $c$ 's service time  $x$  by producing the final regression model  $RG_{ST}(\mathbf{U})$  that takes a weighted combination of all the four models:

$$RG_{ST}(\mathbf{U}) = \frac{\sum_{i=1}^4 (w_{sr_i} \times RG_{U_{sr_i}})}{\sum_{i=1}^4 w_{sr_i}} \quad (1)$$

where the resource contention vector  $\mathbf{U} = \{U_{core}, U_{cache}, U_{diskBW}, U_{networkBW}\}$ .

### B. Extended Performance Model

The extended performance model further employs the queueing system to estimate individual component latency under different request arrival rates. Typically, a queueing system can be described as an  $A/X/m$ , where  $A$  represents the distribution of interarrival time of requests;  $X$  denotes the distribution of service time; and  $m$  is the number of servers. The choice of M/G/1 queueing system is based on the assumption that the distribution of interarrival time of incoming requests are determined by a Poisson process (M for Markov); a component is modeled as a server in the queueing system and the distribution of its service time can follow arbitrary distributions (G for General). Let  $\lambda$  be the monitored request arrival rate and  $\mu$  be the service rate. Let  $\bar{x}$  be the mean service time ( $\bar{x}=1/\mu$ ), and  $var(x)$  be the variance of service time. Component  $c$ 's expected latency  $l$  is calculated as:

$$l = \bar{x} + \frac{\lambda(1 + C_x^2)}{2\mu^2(1 - \rho)} \quad (2)$$

where  $C_x^2 = \frac{var(x)}{\bar{x}^2}$  is the squared coefficient of variation of service time  $x$  and  $\rho = \frac{\lambda}{\mu}$  is the server utilization. In many service components, when the service time follows the exponential distribution, that is, the squared coefficient of variation  $C_x^2 = 1$ , the M/G/1 queueing system equals the M/M/1 queueing system and the expected latency  $l = \frac{1}{\mu - \lambda}$ . At each scheduling interval, a set of resource contention vectors can be collected for each component. By substituting them into Equation 1, the component's corresponding service time  $x$  can be estimated, so its mean and variance can be calculated.

Furthermore, the model computes the overall latency of a service based on its implementation topology. In the online services studied in this work, the processing of a request includes several sequential stages, and each stage parallelizes requests across one or multiple components to aggregate their responses. Hence the calculation of an overall service latency consists of two steps. The first step computes the latency of each stage. Suppose a stage consists of  $C$  parallel components, its latency is the maximum value of these component latencies:

$$l_{stage} = \max_{1 \leq i \leq C} \{l_i\} \quad (3)$$

where  $l_i$  is the latency of component  $c_i$  ( $i=1, \dots, C$ ).

Suppose the service consists of  $S$  sequential stages, the second steps calculates its overall latency:

$$l_{overall} = \sum_{j=1}^S l_{stage_j} \quad (4)$$

where  $l_{stage_j}$  is the latency of the  $j$ th stage ( $j= 1,...,S$ ).

### C. Performance Matrix

Suppose  $m$  components of a service are deployed in  $k$  nodes, the  $m \times k$  performance matrix  $\mathbf{L}$  is constructed using components as rows and nodes as columns. An entry  $\mathbf{L}[i][j]$  denotes the changes in the *overall service latency*  $l_{overall}$  when a component  $c_i$  is migrated from its current node  $n_{current}$  to node  $n_j$  ( $1 \leq i \leq m$  and  $1 \leq j \leq k$ ). This migration may influence all  $m$  components' contention vectors. For any component  $c$  of the service, let its original resource contention vector be  $\mathbf{U}$  and the updated contention vector after the migration be  $\mathbf{U}'$ . Let the resource contention from  $c_i$  itself be  $\mathbf{U}_{c_i}$  and the resource consumption from all programs on node  $n_j$  be  $\mathbf{U}_{n_j}$ . Four situations needs to be considered when calculating the updated contention vector  $\mathbf{U}'$ , as listed in Table III.

Table III  
CALCULATION OF THE UPDATED CONTENTION VECTOR  $\mathbf{U}'$

Type of component $c$	Updated contention vector $\mathbf{U}'$
$c_i$	$\mathbf{U}_{n_j}$
Any component on $n_{current}$	$\mathbf{U} - \mathbf{U}_{c_i}$
Any component on $n_j$	$\mathbf{U} + \mathbf{U}_{c_i}$
Any other component	$\mathbf{U}$

By substituting  $\mathbf{U}'$  into Equations 1 and 2,  $c$ 's updated latency  $l'_i$  can be calculated. We have: (i)  $c_i$ 's latency  $l'_i$  decreases if  $n_j$  has lighter resource contention than  $n_{current}$ ; otherwise  $l'_i$  increases. (ii) All the components on node  $n_{current}$  have decreased latencies because the removal of  $c_i$  alleviates the resource contention on  $n_{current}$ . (iii) All the components on node  $n_j$  have increased latencies because the addition of  $c_i$  aggravates the resource contention on  $n_j$ . (iv) the latencies of other components keep unchanged. Furthermore, by substituting the updated latencies of all  $m$  components into Equations 3 and 4, the updated overall service latency  $l'_{overall}$  can be calculated. Let the overall latency be  $l_{overall}$  before the migration. The entry  $\mathbf{L}[i][j]$  can be calculated as:

$$\mathbf{L}[i][j] = l_{overall} - l'_{overall} \quad (5)$$

Figure 3 shows an example service with three stages, where stage 2 is parallelized into two components  $c_2$  and  $c_3$ . After  $c_2$  is migrated from node  $n_2$  to  $n_4$ ,  $c_4$ 's latency  $l_4$  increases and  $c_2$ 's latency  $l_2$  decreases. By considering all the updated latencies, the overall service latency before and after the migration can be calculated:  $l_{overall}=57\text{ms}$  and  $l'_{overall}=39\text{ms}$ . Hence the reduced latency  $\mathbf{L}[2][4]=18\text{ms}$ .

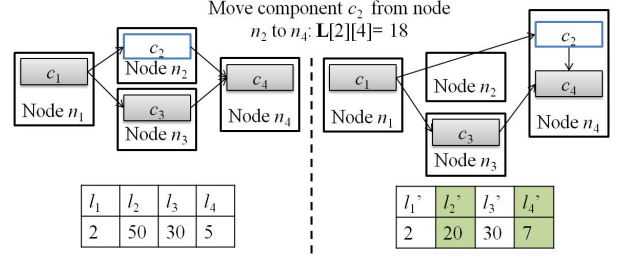


Figure 3. An example of entry  $\mathbf{L}[2][4]$  in the performance matrix

### V. THE COMPONENT-LEVEL SCHEDULING ALGORITHM

Based on the performance matrix of a service, the scheduler can conduct component-node allocations to minimize the overall service latency. Let the  $m$  components  $\{c_1, \dots, c_m\}$  be deployed in  $k$  nodes  $\{n_1, \dots, n_k\}$ , a naïve approach needs a time complexity  $O(k^m)$  to identify the optimal component-node allocation and such exhaustive search is not scalable in practical scenarios. However, performance interference is changing overtime, hence optimizing component-node allocation for a particular dynamic scheduling is not worthwhile. The proposed approach, therefore, applies a greedy algorithm with polynomial computation complexity and the algorithm has several iterations. At each iteration, the algorithm aims to minimize the overall service latency by evaluating all possible component-node migrations and selecting the migration that would reduce the latency the most.

The pseudocode of the algorithm is presented in Algorithm 1. At each scheduling interval, the algorithm first constructs the performance matrix  $\mathbf{L}$  using the performance mode and the monitoring information (line 2). The initial candidate array  $C[N_s]$  takes all  $m$  components as its elements (line 3). The scheduling process then iteratively executes under two conditions: (a)  $C[N_c]$  is not empty; (b) at least one component in  $C[N_c]$  can be migrated (line 5). The second condition indicates that a migration is enforced only when the predicted maximum reduced overall latency  $l_{max}$  is larger than a specified threshold  $\epsilon$ . This threshold prevents inefficient migrations such that the reduced latency cannot compensate the migration cost. In each loop (line 5 to 15), the algorithm first traverses the matrix  $\mathbf{L}$  to identify a set  $S_L$  of entries with the largest value (line 6). Any of these entries denotes the migration of a component that brings the maximal reduction in the overall service latency. If set  $S_L$  contains multiple entries, the algorithm further searches  $S_L$  to find the entry  $\mathbf{L}[c_{max}][n_{Destination}]$  representing the migration that brings the largest reduction to the latency of the migrated component itself (line 7). Component  $c_{c_{max}}$  is regarded as the straggling component and it is allocated to node  $n_{n_{Destination}}$  (line 11). The migrated component  $c_{c_{max}}$  is then removed from the candidate array  $C[N_c]$  and the matrix  $\mathbf{L}$  is updated after this migration.

---

**Algorithm 1** Predictive Component-level Scheduling
 

---

**Require:**  $m$ : the number of components;

$k$ : the number of nodes;

$N_c$ : the number of candidate components to be migrated;

$C[N_c]$ : the index array of candidate components;

$\varepsilon$ : the migration threshold;

$A[m]$ : the component-node allocation array, where  $A[i]$  represents the index of the  $i$ th component's hosting node;

$c_{max}$ : the index of the straggling component  $c_{cmax}$ ;

$n_{Origin}$ : the index of  $c_{cmax}$ 's original node;

$n_{Destination}$ : the index of  $c_{cmax}$ 's destination node;

$l_{i,j}^c$ : A component  $c$ 's reduced latency when migrating from node  $n_i$  to  $n_j$ .

1. Obtain the monitoring information once every scheduling interval;
  2. Construct the performance matrix  $\mathbf{L}$ ;
  3.  $C[N_c] = \{c_1, \dots, c_m\}$ ;
  4.  $l_{max} = \varepsilon + 1$ ;
  5. **while** ( $C[N_c]$  is not empty and  $l_{max} > \varepsilon$ ) **do**
  6. Find a set of entries  $S_L$  in the performance matrix  $\mathbf{L}$  with the largest value;
  7. Find the entry  $\mathbf{L}[c_{max}][n_{Destination}]$  in  $S_L$  with the largest value  $l_{n_{Origin}, n_{Destination}}^{c_{max}}$ ;
  8.  $l_{max} = \mathbf{L}[c_{max}][n_{Destination}]$ ;
  9. **if** ( $l_{max} > \varepsilon$ ) **then**
  10.  $n_{Origin} = A[c_{max}]$ ;
  11.  $A[c_{max}] = n_{Destination}$ ;
  12. Remove  $c_{cmax}$  from  $C[N_c]$ ;
  13. UpdateMatrix( $\mathbf{L}$ ,  $C[N_c]$ ,  $A[m]$ ,  $n_{Origin}$ ,  $n_{Destination}$ );
  14. **end if**
  15. **end while**
  16. Enforce component-node allocation based on  $A[m]$ .
- 

The detailed matrix updating function is given in Algorithm 2. The migration of component  $c_{cmax}$  from node  $n_{nOrigin}$  to  $n_{nDestination}$  alleviates the resource contention on  $n_{nOrigin}$  but aggravates the resource contention on  $n_{nDestination}$ , hence has a twofold impact on the predicted reduction of the overall latency for the following migrations. First of all, components to *migrate to*  $n_{nOrigin}$  ( $n_{nDestination}$ ) have *increased* (*decreased*) reductions in the overall latency. Hence the entries in the  $n_{Origin}$ th and  $n_{Destination}$ th columns should be updated according to Equations 1 to 5 (line 1 to 5). Secondly, components to *migrate out of*  $n_{nOrigin}$  ( $n_{nDestination}$ ) have *decreased* (*increased*) reductions in the overall latency. Each of such components is hosted on either  $n_{nOrigin}$  or  $n_{nDestination}$  (line 3) and the component corresponds to one row in the matrix, hence the entries in this row should be updated (line 7 to 10). Note that component  $c_{cmax}$  is removed from the

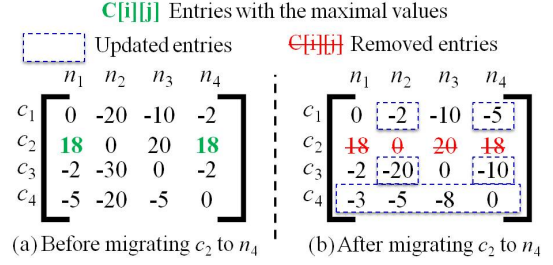


Figure 4. An example loop of migrating component  $c_2$  to node  $n_4$

candidate array  $C[N_c]$ , so all the entries related to  $c_{cmax}$  are not updated.

---

**Algorithm 2** UpdateMatrix( $\mathbf{L}$ ,  $C[N_c]$ ,  $A[m]$ ,  $n_{Origin}$ ,  $n_{Destination}$ )
 

---

**Require:**  $N_r$ : the number of rows to be updated;

$R[N_r]$ : the index array of rows;

1. **for** ( $i=0$ ;  $i < m$ ;  $i++$ ) **do**
  2. Update  $\mathbf{L}[i][n_{Origin}]$  and  $\mathbf{L}[i][n_{Destination}]$ ;
  3. **if** ( $(A[i] == n_{Origin} \text{ or } A[i] == n_{Destination})$  and  $c_i \in C[N_c]$ ) **then**
  4. Add the  $i$ th row  $r_i$  to  $R[N_r]$ ;
  5. **end if**
  6. **end for**
  7. **for each** row  $r_j$  in  $R[N_r]$  **do**
  8. **for** ( $v=0$ ;  $v < k$ ;  $v++$ ) **do**
  9. Update  $\mathbf{L}[j][v]$ ;
  10. **end for**
  11. **end for**
- 

Figure 4 illustrates an example loop, in which migrating component  $c_2$  to either node  $n_1$  or  $n_4$  can result in the maximal reduction in the overall service latency. At the same time, the reduction in  $c_2$ 's latency is 20ms when it is migrated to  $n_1$  and 30ms when it is migrated to  $n_4$ , which indicates  $c_2$  suffers from less performance interference when it is hosted on  $n_4$ . Hence the scheduling algorithm allocates  $c_2$  to  $n_4$ , after which the entries in the second and fourth columns (representing components to migrate to nodes  $n_2$  and  $n_4$ ), and the fourth row (representing components to migrate out of  $n_2$  and  $n_4$ ) are updated. All the entries in the second row are not updated because  $c_2$  is not considered in the following scheduling. Let the migration threshold be  $\varepsilon=5$ ms, we can see that after this loop, the scheduling process is completed because no further effective migration can be conducted: all the values of entries in the updated matrix are smaller than 5ms.

The complexity of each scheduling interval is  $O(m^2 \cdot k)$ . Specifically, the performance matrix can be constructed in  $O(m \cdot k)$  time. The scheduling process can be completed within  $m$  loops, where each loops takes  $O(m \cdot k)$  to find the optimal migration and  $O(m + m \cdot k)$  to update the matrix.

## VI. EVALUATION

### A. Experiment methodology

**Experiment platform.** The experiments were conducted in a set of 30 nodes connected with a 1Gb ethernet network. Each node has two 6-core Xeon E5645 processors and hosts multiple VMs using Xen Virtual Machine Monitor (VMM). The operating system of both physical machines and VMs is SUSE Linux Enterprise Server (SLES)-11-SP1. The Xen, JDK versions are 4.0, 1.7.0, respectively. In addition, the versions of Nutch (search engine), Hadoop, and Spark are 1.1, 1.0.2, and 0.8.0, and the versions of Storm, Python, and Zookeeper are 0.9.2, 2.6, and 3.4.6, respectively.

**Workloads.** We use representative workloads from the open-source BigDataBench workload suite [3]. The Nutch web search engine [1] represents the latency-critical online service and its online web search performance was tested. As shown in Figure 1, this service has three stages and we call the components at Stage 1, 2, and 3 *segmenting components*, *searching components*, and *aggregating components*, respectively. The batch jobs involve a variety of Hadoop MapReduce and Spark jobs. Hadoop jobs include the two typical CPU-intensive workloads with float point and integer calculations (Naïve Bayes classification and WordCount) and one workload having similar demands for CPU and I/O resources (Page Index). Spark jobs are mostly I/O-intensive workloads including Naïve Bayes, WordCount and Sort. These short-running batch jobs whose execution time ranges from a few seconds to several minutes represent a large fraction of jobs in today’s data center workloads [13], [24].

**Compared techniques.** Two classes of state-of-the-art latency reduction techniques are compared. (i) *Request redundancy* [11], [26], [27]. For each request, multiple replicas are created for parallel execution and only the quickest replica is used. Two different redundancy policies, which generate three or five replicas were tested. (ii) *Request reissue* [14], [18]. A request is first sent to the most approximate component for execution, and a replica of this request is sent if the first one is not completed after a brief delay. The quickest replica is then used. Two reissue policies, which send a secondary request after the first has been executed for more than the 90th percentile or the 99th percentile of the expected latency for this class of requests, were tested.

For simplicity, we will call the four compared techniques, *RED-3*, *RED-5*, *RI-90*, *RI-99*. We also call the basic technique without any redundancy or reissue *Basic*, and our predictive component-level scheduling approach *PCS*.

**Metrics.** Two metrics are used to evaluate the performance of the search engine service. The *first* metric is the 99th percentile latency of individual components of all requests. In the case of the request redundancy and reissue techniques, this metric denotes latencies of components belonging to the quickest replica. The *second* metric is the

average overall service latency of all requests.

**Measurement method.** In the experiments, the monitor dynamically inspects the running service, including its request arrival rate and resource contention information listed in Table II. The monitor obtains the request arrival rate and the system-level contention information once every second and the micro-architectural contention information once every minute. This measurement method guarantees low overheads in monitoring and does not affect the application performance.

### B. Prediction accuracy

The effectiveness of the proposed scheduling approach is considerably impacted by the performance model’s accuracy. To evaluate this accuracy, we ran each searching component of the service on a VM with 1 core and 1GB memory, and used another VM with 4 core and 4GB memory co-located on the same node to run a Hadoop or Spark job of different input sizes. In each test, we trained the regression models based on the historical running information and predicted the component’s service using the constructed models.

As listed in Figure 5, in our evaluation, the Hadoop workloads have 20 different input sizes ranging from 50MB to 4GB, and the Spark workloads have 10 different input sizes ranging from 200MB to 7GB, thus having distinct performance interferences to the component’s latency. As shown in Figure 5, the prediction errors are *smaller* than 3%, 5%, and 8% in 63.33%, 82.22%, and 96.67% of the evaluation cases, respectively. When considering all the input sizes, the average prediction error is 2.68%, indicating the performance model keeps a good track of the observed latency and it is sufficient for our scheduling heuristic to achieve a near-optimal performance.

### C. Service Performance

**Evaluation setting.** Following the deployment settings of the previous section, we tested the performance of the Nutch search engine service whose searching components are deployed in 100 VMs. Each component co-locates with a mixed of batch jobs running on VMs of the same node. The Hadoop workloads were tested with continuously changing input data sizes ranging from 1MB to 10GB. Six request arrival rates, namely 10, 20, 50, 100, 200, 500 requests/second, were tested to compare the latency reduction techniques under online services’ diurnal variation in load.

**Migration threshold.** As explained in Section V, the proposed scheduling algorithm employs a migration threshold to control latency reduction and throttle non-beneficial component migration. This threshold should be reasonably high to filter out most of the detrimental migrations whose overheads are larger than the possible latency reduction. On the other hand, the threshold cannot be too high to miss the opportunities for latency reduction. The major overhead of migrating a component is caused by the movement from its



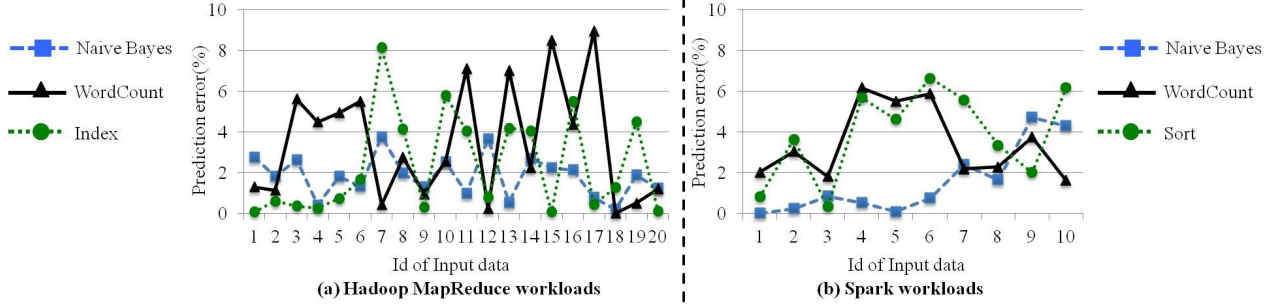


Figure 5. Prediction errors of the performance model under different performance interferences

current VM to the destination VM. The component runs on a VM installed Storm and its migration is enforced by calling Storm’s deployment APIs. Specifically, Storm first uploads the source codes (e.g. codes for looking up indexes for documents) and the configuration information of the component to ZooKeeper [9], a widely used distributed coordination system to manage application deployment. ZooKeeper then allocates them to a new component on the destination VM. At each scheduling interval, the migration of components (e.g. 10 to 20 components) can be completed within 3 seconds without interrupting the running services and only causes small consumptions of memory and I/O resources. Considering the migration cost, we find out that 5% of the accepted overall service latency (100ms) is a reasonable threshold value for the studied online services and thus the threshold in scheduling is set as 5ms. Applying an adaptive threshold to improve the service performance is possible, but it is beyond the scope of this paper.

**Evaluation results.** Figure 6 shows the comparison of service performance for six different techniques. The results show that PCS achieves the smallest tail latencies and the overall service latencies in all cases. This is because during the execution of the service, PCS dynamically enforces different component-node allocations along with the latest performance interference changes on different nodes and reduces the component latency variability by migrating the straggling components to nodes with less resource contentions.

By contrast, the *request redundancy* technique just collects responses from the quickest component based on the current service deployment, missing the opportunity to migrate the components to the idle nodes with the least performance interference. Figure 6(a) and (b) show that this technique achieves some latency reduction under light workloads. However, when the arrival rate gradually increases to 500, Figure 6(c) to (f) show that this technique adversely causes longer latencies compared to those of Basic. In particular, RED-5 causes the longest latencies because it produces the largest workloads, namely incurring the longest queueing delay, among all techniques. Although

the redundancy technique employs the cancelation mechanism that sends messages to cancel other queuing replicas when one replica begins execution, the components still execute replicas of the same request unnecessarily. This phenomenon mainly comes from two sources: (i) all replicas of a request are sent to multiple components simultaneously, hence two components having similar performances may start executing the requests in similar time; (ii) there is a network message delay for different components to communicate each other’s status, hence two components may start executing the same request and the cancelation messages are both in the flight to each other. Moreover, the *request reissue* technique applies a conservative redundancy mechanism that only creates replicas for requests judged as outliers (i.e. requests whose execution time is larger than an expected latency). Results in Figure 6 show that compared to the request redundancy technique, this conservative reissue technique causes less performance deterioration when load becomes heavier.

**Results.** Considering all the six request arrival rates, PCS achieves 67.05% reduction in the 99th component latency and 64.16% reduction in the overall service latency when comparing to the request redundancy and reissue techniques.

#### D. Scalability of scheduling

In proposed scheduling heuristic, the used performance mode is constructed based on profiling of each component. That is, only one out of all homogeneous components needs to be profiled and thereby avoiding the scalability issue associated with the service profiling. For example, in the tested search engine service, only three components (segmenting, searching and aggregating) need to be profiled. Meanwhile, the proposed scheduling algorithm estimates the service performance by analyzing the resource contention information obtained from each component, and hence the *analysis time* scales only linearly with the number of components. Another important aspect of the scalability of scheduling is to *search* the appropriate component-code allocation, and the time complexity of this is  $O(m^2 \cdot k)$  when allocating  $m$  components to  $k$  nodes.



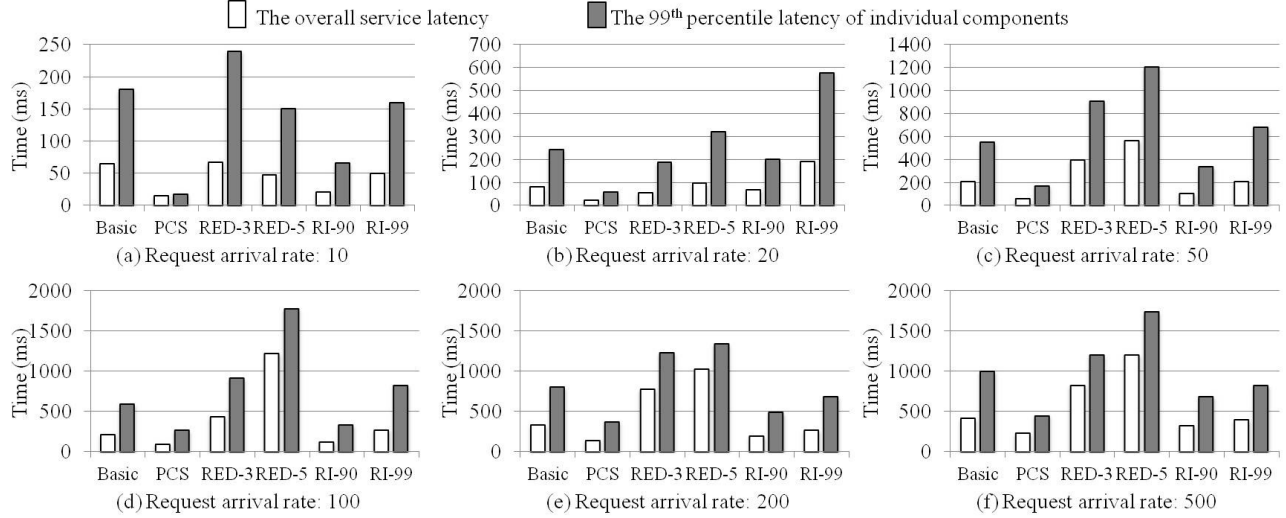


Figure 6. Comparison of overall service latency and the tail latency under different request arrival rates

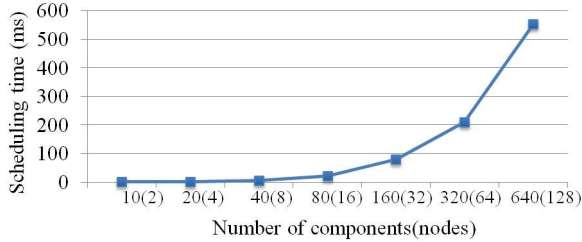


Figure 7. Scalability of the scheduling algorithm

To evaluate the scheduling algorithm scalability, we measured both the analysis and searching time under different numbers of components and nodes. Figure 7 shows that even if the number of components reaches 640 (and the number of nodes reaches 128), the algorithm takes only 551ms to complete. This time is less than 0.1% of the 600 seconds scheduling interval and hence can be ignored. For services with more components, the scheduler could apply a hierarchical strategy that divides the components into small groups of 640 components or less and finds the appropriate component-node allocation between groups and then within groups. The scheduling overhead therefore can remain low even with a large number of components.

## VII. RELATED WORK

### A. Application-level management of service performance

At present, two categories of techniques have been proposed to meet the performance requirement of latency-critical services by alleviate the performance degradation due to resource sharing and contention. The *first category* of techniques disallow the co-location of services with applications incurring large contention of resources such

as caches [22] and CPU resources [30]. The *second category* of techniques dynamically manage applications to meet their performance requirement at run-time according to the monitored interference metrics, such as the LLC miss rate reflecting cache contentions [10] and the bandwidths reflecting I/O resource contentions [29]. These techniques focus on addressing performance variability of applications by viewing the application as a whole, ignoring issues relating to fine-grained latency variability of its individual components. However, these components' tail latency dominates performance of large-scale, parallel services.

### B. Tail latency reduction techniques

We now review four categories of reduction techniques.

**Modifying hardware/software systems.** These techniques aim at solving the tail latency caused by system design issues. Those include architecture-level design that disables the power saving model to promote system performance [28]; OS-level design that changes the default kernel scheduler to a better scheduler (e.g. Borrowed Virtual Time (BVT)) with better support for time-sensitive requests [23].

**Adding additional resources.** These techniques require additional resources to handle slow requests, either by increasing the parallelism degree of the request processing [19] or adding new server components [26].

**Partially processing request.** These techniques reduce tail latency by only using a portion (e.g. 90%) of the quickest sub-requests [18] or a synopsis representing the entire input data at a high level of approximation [17], thus sacrificing result correctness such as query accuracy for reducing service latency.

The approach proposed in this work can work together with the above techniques to reduce tail latency, thus forming a complement to these techniques. Both this work and the

fourth category of techniques, namely **request redundancy** [11], [26], [27] and **reissue** [14], [18] explained in Section VI-A, reduce tail latency by addressing component latency variability. The key idea of the request redundancy technique is to execute the same request on multiple components so as to reduce its latency by using the quickest one. Although these techniques work well when workloads underutilize system resources [26], they start hurting the service performance and adversely worsen the latency when load gets heavier [25].

### VIII. CONCLUSION

This paper presents a component-level scheduling framework that can dynamically schedule components of a service across hundreds of machines in a cloud data center. To adapt to the changing performance interferences and workloads, this framework leverages cost-efficient online monitors and an analytic performance model to simultaneously predict the components' latency when running on different nodes. Using the predicted performance, the scheduler identifies straggling components and enforces near optimum component-node allocations. By comparing to the best well-known techniques on reducing tail latency, we demonstrate that our approach achieves significant reductions in both component tail latency and overall service latency.

### IX. ACKNOWLEDGEMENTS

We sincerely thank Moustafa M. Ghanem and Li Guo and for their useful comments, and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work is supported by Chinese 973 projects under Grants No. 2014CB340402.

### REFERENCES

- [1] Apache nutch search. <http://nutch.apache.org/>.
- [2] Apache storm. <https://storm.apache.org/>.
- [3] Bigdatabench. <http://prof.ict.ac.cn/BigDataBench/>.
- [4] Drill. <https://drill.apache.org/>.
- [5] Hadoop. <http://hadoop.apache.org/>.
- [6] Oprofile. <http://oprofile.sourceforge.net/>.
- [7] Perf. <https://perf.wiki.kernel.org/>.
- [8] Spark. <http://spark.apache.org/>.
- [9] Zookeeper. <http://zookeeper.apache.org/>.
- [10] Jeongseob Ahn, Changdae Kim, Jaewung Han, Young-ri Choi, and Jaehyuk Huh. Dynamic virtual machine scheduling in clouds for architectural shared resources. *HotCloud'12*, 2012.
- [11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: attack of the clones. In *NSDI'13*, pages 185–198, 2013.
- [12] Rodrigo N Calheiros, Rajiv Ranjan, and Rajkumar Buyya. Virtual machine provisioning based on analytical performance and qos in cloud computing environments. In *ICPP'11*, pages 295–304. IEEE, 2011.
- [13] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.
- [14] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
- [15] Rui Han, Moustafa M Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32:82–98, 2014.
- [16] Rui Han, Li Guo, Moustafa M Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *CCGrid'12*, pages 644–651. IEEE, 2012.
- [17] Rui Han, Junwei Wang, Fengming Ge, Jose Luis Vazquez-Poletti, and Jianfeng Zhan. Sarp: producing approximate results with small correctness losses for cloud interactive services. In *CF'15*, page 22. ACM, 2015.
- [18] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *SIGCOMM'13*, pages 219–230. ACM, 2013.
- [19] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Predictive parallelization: Taming tail latencies in web search. In *SIGIR'14*, pages 253–262. ACM, 2014.
- [20] Zhen Jia, Jianfeng Zhan, Wang Lei, Rui Han, and Sally A. McKee. Characterizing and subsetting big data workloads. In *IISWC'14*. IEEE, 2014.
- [21] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: predictable low latency for data center applications. In *SoCC'12*, page 9. ACM, 2012.
- [22] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *ASPLOS'14*, pages 729–742. ACM, 2014.
- [23] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EuroSys'14*, page 4. ACM, 2014.
- [24] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC'12*, pages 7–19. ACM, 2012.
- [25] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. When do redundant requests reduce latency? Technical report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2013.
- [26] Christopher Stewart, Aniket Chakrabarti, and Rean Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *ICAC'13*, pages 265–277. USENIX, 2013.
- [27] Ashish Vulimiri, Oliver Michel, P Godfrey, and Scott Shenker. More is less: reducing latency via redundancy. In *HotNets'12*, pages 13–18. ACM, 2012.
- [28] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien An Lai, Masazumi Matsubara, and Calton Pu. Impact of dvfs on n-tier application performance. In *TRIOS'13*, page 5, 2013.
- [29] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *PACT'10*, pages 237–248. ACM, 2010.
- [30] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: avoiding long tails in the cloud. In *NSDI'13*, pages 329–342, 2013.
- [31] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA'13*, pages 607–618. ACM, 2013.